COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055

THE STANFORD ADA STYLE CHECKER: AN APPLICATION OF THE ANNA TOOLS AND METHODOLOGY

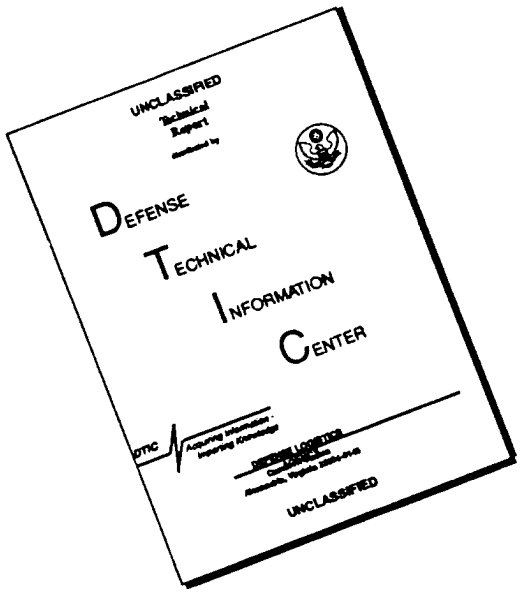
Michał Walicki
Jens Ulrik Skakkebæk
Sriram Sankar

Technical Report: **CSL–TR–91–488**
(Program Analysis and Verification Group Report No. 55)

August, 1991

Michał Walicki was on leave from the University of Bergen, Norway, and was supported by the Royal Norwegian Council for Scientific and Industrial Research. Jens Ulrik Skakkebæk was on leave from the Technical University of Denmark, Denmark. Sriram Sankar was supported by the Defense Advanced Research Projects Agency contracts N00039-84-C-0211 and N00039-91-C-0162.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

THE STANFORD ADA STYLE CHECKER: AN APPLICATION OF THE ANNA TOOLS AND METHODOLOGY

Michał Walicki
Jens Ulrik Skakkebæk
Sriram Sankar

Technical Report: **CSL–TR–91–488**
(Program Analysis and Verification Group Report No. 55)

August, 1991

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California - 94305

Abstract

This report describes the *Ada style checker*, which was designed and constructed in Winter and Spring 1989–9 The style checker is based on the Stanford Anna Tools and has been annotated using Anna. The style check examines Ada programs for "correct style" which is defined in a *style specification language (SSL)*. A *style check generator* is used to automatically generate a style checker based on a set of style specifications.

Keywords—*Ada, Anna, style specification, style checking.*

Contents

Chapter 1

Introduction

Overview. This report describes the *Ada style checker*, which was designed and constructed in Winter and Spring 1989–90. The style checker is based on the Stanford Anna Tools and has been annotated using Anna [LvHBO87]. The style checker examines Ada [Ada83] programs for "correct style" which is defined in a *style specification language (SSL)*. Hence, the Ada style checker takes as input an Ada program and a set of style specifications and produces as output a list of style violations in the Ada program. An example of a style specification is:

Use clauses may not be used

The output of the style checker for this example will be an error message corresponding to each occurrence of a use clause in the input Ada program. Figure 1.1 illustrates the operation of the Ada style checker.

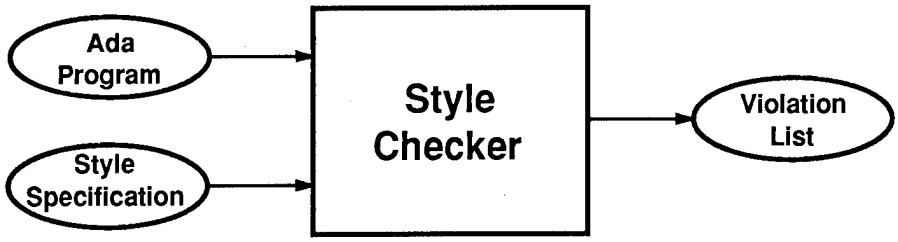


Figure 1.1: The I/O specification of the style checker

Use of the style checker. Style checking of programs is a typical activity in software houses where large volumes of software are written regularly. Administrators or managers in these software houses set out programming style guidelines for their programmers. Typically, these guidelines are English documents.

We refer to the writer of the style guidelines as the *system manager*, someone with a high-level understanding of software development, but who does not necessarily write any software themselves.

The style specification language. Since we are automating the process of style checking, the style specifications have to be written out in some formal, machine processable manner. Also, given the qualifications of the system manager, we require a high-level approach to writing style specifications. Our style specification

language satisfies both these requirements. The style specification language provides the capability of defining style guidelines using a predefined vocabulary of *concepts*. The style specification of use clauses shown earlier can be written out in our style specification language as:

$$count(use\_clause) = 0$$

or as:

$$no(use\_clause)$$

Details on interpreting such specifications are provided later.

The style checker generator. We are faced with the often expensive task of processing the high-level style specification into an internal form before an Ada program can be compared to it. Given the typical scenario in which the style checker is used, we can assume that the style specifications are set forth by the system manager once and for all, and that based on them, many programs are then written and checked for style violations. The original style specifications are seldom changed. Hence, our approach is to process the style specifications only when they are changed and to generate a style checker based on these specifications. This generated style checker is then used to examine Ada programs for style violations. Given a fixed set of style specifications, this process allows for a much faster style checker. Figure 1.2 illustrates this approach. A Style Specification is first written and given as input to the SSL Parser and then to the Style Checker Generator. This produces the executable Style Checker. The Style Checker can now be used to check Ada programs for style violations.

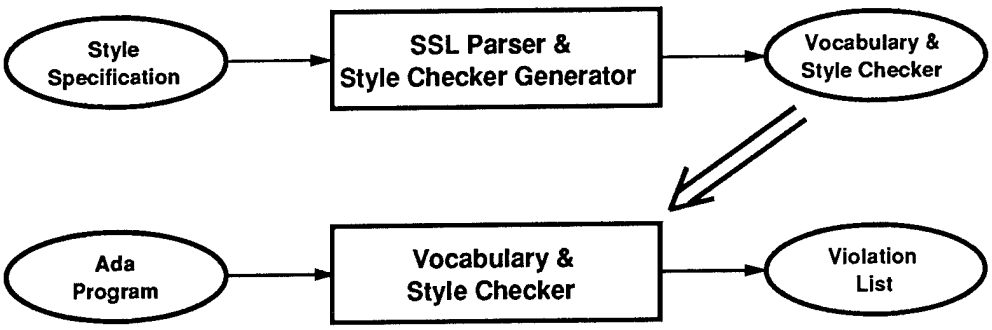


Figure 1.2: The style checker generator

The Style Checker Generator converts the high-level style specification into an Ada program with Anna constraints. This generated Ada program takes the Ada program on which style guidelines need to be checked as input. The input program is parsed and semanticized, resulting in a DIANA [EBGW83] *Abstract Syntax Tree* (AST) whose nodes are decorated with semantic information. The generated Ada program then invokes a routine that traverses the complete DIANA AST.

The generated Anna constraints are annotations on this tree traversal routine. They specify what the tree traversal routine is to expect while traversing the DIANA AST. Anything unexpected corresponds to a style guideline violation.

The final step in generating the style checker is to apply the Anna Transformer [San89] to the Anna/Ada program generated by the Style Checker Generator. This results in an Ada program where the Anna constraints are transformed into Ada checks on the Ada program. Violation of these checks cause an Anna constraint violation to be reported. Hence, when this transformed program is executed on an input Ada program, Anna constraint

violations occur every time a style guideline violation is detected. The Anna constraint violations are handled in a manner appropriate for reporting style guideline violations, hence the user of the style checker is unaware of the Anna constraint violations.

The Anna constraints may refer to various parts of the Ada program by using predefined *language objects* (*e.g.*, *use_clause*). Furthermore, calls to various predefined *operations* on these language objects are allowed (*e.g.*, *count*, *no*). The language objects and the operations are parts of the SSL *vocabulary*.

The language objects and operations are represented in Ada as discrete types and functions, which are contained in a *vocabulary package*. The Anna constraints thus make use of these types and contain calls to the functions.

The style checker may be activated from the command line (using the command line interface provided by the system) or from another Ada program (using the programmatic interface). The user specifies the name of an input Ada program to be checked when activating the style checker. When the input program has been completed checked, the list of style guideline violations is returned to the calling routine.

Implementation status. The Style Checker Generator has not been implemented. However, a set of style specifications based on the SPC software guide [SC88] have been hand-translated to produce a style checker for these specifications. This version is called the *Stanford Ada Style Checker*. This style checker has been used on many practical Ada programs.

Organization of this report. Chapter 2 is a description of the SSL. Chapter 3 describes the overall structure of the system. A user manual for the present version of the style checker can be found in Chapter 4 and 5. Chapter 4 describes the command line interface for invoking the style checker, and Chapter 5 describes the programmatic interface. Chapter 6 concludes with suggestions for future work — the next steps in the development of components such as the SSL compiler and the style checker generator.

Appendix A is a detailed description of the style guidelines inserted in the current version of the style checker. A maintenance guide can be found in Appendix B.

1.1 Acknowledgements

We wish to thank Prof. David C. Luckham for his inspiration and support. We are also grateful to Steve Sherman and Barry Schiff of the Lockheed Missiles and Space Corporation for introducing us to the idea of a general framework for style checking, and for sharing their experiences as software managers at Lockheed. We are thankful to members of Stanford University's Program Analysis and Verification Group, and especially to Geoff Mendal for his assistance in adapting the Anna tools for the style checker.

Chapter 2

The Style Specification Language

We begin by describing the interface language for specifying style constraints — *the style specification language (SSL)*. We hope that the description of SSL can give the reader a high-level understanding of the system needed by the system manager — an idea of the intended functionality and of the motivations for its various parts.

SSL is a language for describing both the syntactic and the semantic properties of programs. It may be thought of as consisting of sentences which we will call the *style guidelines*. A style guideline is specified by selecting three types of arguments:

- the language objects which are to be constrained (*e.g.*, *procedure_declaration, variable, if_statement*; these are called object *types*),
- the context which is relevant for the constraint (*e.g.*, *in function*, meaning that one wants to constrain only the objects appearing within functions; the default context being the whole program),
- the actual constraint, that is, a boolean expression over some objects and operations on these objects.

The first two kinds of arguments relate directly to the language objects, the last to their properties. The object types and their properties may be added whenever functionality of the system needs to be extended.

2.1 SSL Syntax and Semantics

A style specification written out in SSL comprises a sequence of SSL statements. Each statement describes a constraint, and a scope over which this constraint applies. The constraint can be any arbitrary Anna boolean expression. A name (`id`) is provided for each statement, and an optional explanatory string may be provided as an error message to display in the event of a style violation with respect to this statement.

```
Style_Specification ::= { Statement }

Statement ::= Scope : Constraint ( id [ , explanation ] ) ;

Constraint ::= Anna_Boolean_Expression
```

The scope is defined by defining a logical variable and quantifying it in a particular manner. There are two aspects to this quantification: (1) specifying the kind or type of this variable, and (2) specifying where the variable can occur in the program being analyzed.

```
Scope ::= FOR ALL variable : Kind [ Context ]
```

The kind or type of the variable is specified as being any one from a set of language objects.

```
Kind ::= language_object { OR language_object }
```

The context specifies where the variable can occur in the program being analyzed. It is specified by describing the block structure within which the variable can occur — *e.g.*, that it is immediately within a procedure, or that it is immediately within a task which in turn is within a library package. Various different contexts can be specified and these can be combined using the logical operators **AND**, **OR**, and **NOT**.

```
Context ::=   Context OR Context
            | Context AND Context
            | ( Context )
            | NOT Context
            | Single_Context_Def
```

```
Single_Context_Def ::= IN [..] Language_Object { [..|, ] Language_Object } [..]
```

Single_Context_Def defines a particular context. The language objects in this definition have to be entities that define scopes (*e.g.*, procedure body, package specification, etc.). The special symbols (**,** and **..**) have the following meanings:

,: This symbol indicates that the scoping unit to its left is immediately nested within the scoping unit to its right.

..: This symbol indicates that the scoping unit to its left is nested at some arbitrary level within the scoping unit to its right. If the scoping unit to the right of this symbol is omitted, it is considered to be the outermost scope (the library level). Similarly, if the scoping unit to the left of this symbol is omitted, it is considered to be the current scoping unit. The examples below will clarify this.

Examples:

1. **IN function,package_body** :
   Any scope that is immediately within a function, which in turn is immediately within a library package body.
2. **IN procedure,package_body..** :
   Any scope that is immediately within a procedure, which in turn is immediately within a package body, which in turn is nested within an arbitrary number of scoping units.
3. **IN ..function..task_body..** :
   Any scope that is nested somewhere within a function, which in turn is nested somewhere within a task body, which in turn is nested within an arbitrary number of scoping units.

More detailed examples of style specifications written in SSL are provided in Appendix A.3.

2.2 The Vocabulary Package

The Vocabulary Package implements the SSL vocabulary. The functions used in the style specifications reside here together with the language object definitions.

All functions and objects used in the specification are defined here as corresponding Ada functions and objects. Modifying the SSL vocabulary is done by modifying the vocabulary package. For instance, adding an extra function to the vocabulary will require adding an extra Ada function.

This section describes the objects and functions currently implemented.

2.2.1 Language Objects

The language objects represent all the different language constructs found in Ada, such as if-statements, package-bodies, etc. The object names are chosen to resemble the corresponding language constructs as much as possible.

The language objects defined in the vocabulary package are listed in Table 2.1.

package_decl	package_body	generic_package_decl
function_body	entry_call	entry_decl
exception_raise	exception_decl	with
use	constant_decl	deferred_constant_decl
derived_type_decl	private_l_type_decl	private_type_decl
incomplete_type_decl	subtype_decl	var_decl
task_decl	task_body	if
else	elsif	case
loop	goto	abort
pragma	attribute	in_parameter
and	or	main_ob

Table 2.1: Language Objects

In some situations, it is advantageous to talk about classes of language objects rather than the individual language objects themselves. For this purpose, a few *derived objects* have been defined, each of which represents a class of language objects. The derived objects implemented in the vocabulary package are listed in Table 2.2.

nil	a DIANA node which does not correspond to any language construct
any	any language construct
declaration	any type of declaration (object with _decl suffix in its name)
nested_stm	if-, case-, or loop-statement
subprogram_call	function_call or procedure_call
subprogram_decl	function_decl or procedure_decl
subprogram_body	function_body or procedure_body

Table 2.2: Derived Objects

2.2.2 Functions

The functions currently implemented are described below. The parameter *program part* corresponds to the DIANA AST node where the tree traversal routine is currently at.

Function: Count_1
Parameters: *program part*, language object
Returns the number of objects of the specified language object kind at the first nesting level of the program part.

Function: Count_All
Parameters: *program part*, language object
Returns the number of objects of the specified language object kind encountered in the program part.

Function: Self_Nesting
Parameters: *program part*
Returns the number of objects of the same kind as the program part nested within each other.

Function: Object_Nesting
Parameters: *program part*, language object
Returns the number of nested levels within the program part in which the language object occurs.

Function: Logic_Nesting
Parameters: *program part*, language object
The language object denotes a set of compound statements. This function returns the number of levels to which these statements are nested within each other in the given program part.

Function: Named_Stm
Parameters: *program part*
Returns true if the program part is a named statement.

Function: Named_Par_Assoc
Parameters: *program part*
Returns false if the program part is a subprogram call or an entry call and at least one of the actual parameters has no formal parameter name associated with it.

Function: Is_Predefined
Parameters: *program part*
Returns true if program part is defined with package Standard, *i.e.*, the program part denotes a predefined entity.

Function: Refers_Predefined
Parameters: *program part*, predefined type name
Returns true if the program part contains references to the specified predefined type.

Function: With_Default
Parameters: *program part*
Returns false if the program part corresponds to a parameter and has no default value.

Function: Separate_Comp
Parameters: *program part*
Program part has to denote a compilation (a file). This function returns true if either: (1) the compilation contains only bodies (*e.g.*, package bodies, subprogram bodies); or (2) the compilation contains exactly one specification (*e.g.*, a package specification, or a generic unit specification).

Function: Is_Within_Private
Parameters: *program part*
Returns true if the program part is within the private part of a package.

Function: Source_Pos
Parameters: *program part*
This function is used for returning the position of a given language construct in the source code.

Examples of use and more detailed description of these functions can be found in Appendix A.

Chapter 3

System Overview

As described in Chapter 2, the system manager can specify the style guidelines using language objects (which denote some language construct) and operations on these objects. In contrast to the syntax of SSL, the collections of objects and operations are considered to be dynamic entities, which can be modified several times during the life cycle of the style checker. This can occur for instance when new operations are needed in order to express a new style guideline.

This chapter gives the reader a brief overview of the system and shows how it is affected by modifying the objects and the operations.

3.1 The System Structure

The following is a brief high-level description of each element of the system. The numbers refer to the numbers on Figure 3.1.

1. The system manager creates a style specification, using the SSL (1).
2. The parser (2) parses the system manager's style specification based on the BNF grammar described in Chapter 2 and an executable version of style checker is constructed (5,6).
3. This executable image (7) will then take Ada programs as input and produce violation indications as output. This program will depend heavily on the set of Anna Tools used for parsing, semanticizing and manipulating the DIANA representation of the input program (see [EBGW83]).

3.2 Modifications

We consider the collection of objects and the collection of operations to be dynamic entities, *i.e.*, they are subject to change during the lifetime of the system. Because the system structure must allow modifications to take place easily, our system allows modification of the collection of objects or the collection of operations simply by modification of the vocabulary package (3) and the mapping (4). For example, adding an operation can be done in the following way: a function is declared and formally specified in the visible part of the vocabulary package and implemented in the body of the vocabulary package. This will of course require some knowledge of the implementation. A mapping is also defined between the actual function/procedure in the vocabulary package and the operation in the system manager's view. The vocabulary package is then transformed using the Anna

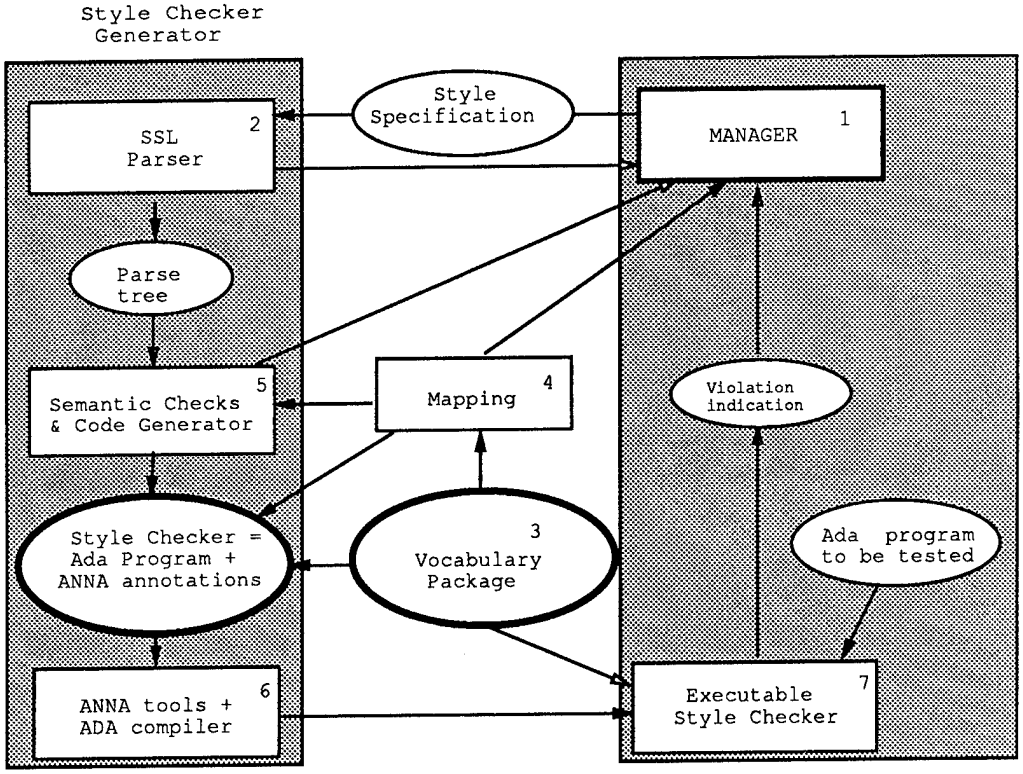


Figure 3.1: System Architecture

tools and is compiled using the Ada compiler to a separate module. At this point, the function is ready for use by the system manager in his style specification.

The functions and procedures in the package can be divided into several layers of abstraction, *i.e.*, functions on higher levels of abstraction can use lower level functions in their implementation. This approach will ease the modifications of the package.

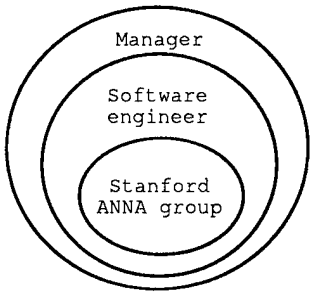


Figure 3.2: Layers of the Vocabulary Package

The system manager and the people maintaining the functions at different levels of abstraction all have a different view (see Figure 3.2). The system manager will view the functions and procedures as operations on the Ada language entities, without knowing anything about the implementation of the system. The system manager's style specification will only be based on knowledge of the Ada language (*e.g.*, notions of nested language constructs, if-then-else statements, package bodies and specifications, and the like).

The person maintaining the the functions and procedures at the highest level of abstraction must have some knowledge about the underlying abstract syntax tree, but need not know the details. A higher level function can be implemented using the low level functions. A system manager would typically get one of his software engineers to maintain these higher level functions and procedures, while maintaining the lower level functions and procedures will require an intimate knowledge of the underlying DIANA tree. An example of people maintaining these is members of the Stanford Anna group or others familiar with the Anna Tools.

Chapter 4

Using The Style Checker

In this chapter, we will show how the style checker can be activated at the command line and provide an example. We assume, that the style checker has been installed in the correct directory, search paths set up, etc.

The style checker is activated by running the program named 'style'.

In order to run, the user has to specify the name of the Ada file to be style checked.

For instance, typing at the command line:

```
style test
```

will cause the style checker to check the compilation unit 'test.a', using the default option values (see below).

However, the user is also allowed to provide options when activating the style checker. The command line format is as follows:

style {-[c|e|v|s|l|n#]} <file name >

where:

-c : *Current compilation unit only, NOT with'ed packages (default false).* If this option is not specified (false), all the packages with'ed in the compilation unit will be checked before the current compilation unit. Note, that this may cause many compilation units to be checked by the style checker, since the with'ed compilation units may also have with'ed packages themselves. **This option is not implemented in the current version. In the version implemented by the date of this publication, only the current compilation unit will be checked.**

-e : *Suppress explanation in output (default false).* If the option is not specified (false), the style violation will only appear with a source position and a name, both on the screen and in the `.vio`-file (see below). The explanation is currently hard-coded into the program. However, the planned style checker generator is intended to include the option of allowing the system manager to specify the explanation himself.

-v : *Verbose mode. Log to screen. (default false).* If this switch is specified (true), output will be sent to the standard output during the style checking, *i.e.*, the violations that are detected will be sent to the output device.

-s : *Continue on semantic error (default is to abort style checking when semantic error occurs).* Semantic checking of the input program is performed before the style checking takes place. If a semantic error occurs, the style checking will halt, since all semantic information needed to perform the style checking properly might not be present. However, some semantic errors might not influence the style checking. A warning will be displayed, if style checking continues despite a semantic error.

-l : *Suppress .vio log file.* The style checker will by default dump a list of violations detected into a file with the postfix '`.vio`'. The user can suppress this by providing this option at the command line.

-n# : *Stop at violation number # (default 0).* If not specified, style checking will be performed until the end of the input file is reached. However, if the user for some reason would like the style checking to stop after a certain number of style violations, this can be specified using this flag. The number 0 will cause the style checker to report all violations detected.

<file name> : *File name with .a omitted.*

Activating the style checker without any parameters at all will produce a listing of the command line format and the options allowed.

4.1 Example

Consider the following package stored in the the file 'ex1.a':

```
package Example1 is
end Example1;

package body Example1 is
   x: Integer;
   b: Boolean;
begin
   b := b and (x = 0);
end Example1;
```

Using the style rules, that have been included in the style checker so far (see Appendix A), activation of the style checker by the command `style -v ex1` yields the following output on the screen:

```
Stanford Ada Style Checker Version 1.20, Rev. 4/30/90

Source file: ex1.a
List   file: ex1.vio
Output file: ex1.sty

Loading parse table . . . . . parse tables loaded
Parsing source file . . . . . parsing complete. No syntax errors.
Checking semantics  . . . . . semantic OK - output in .sem file
Checking source file for style guideline conformance . . .
*** ERROR: STYLE VIOLATION Line  1, Column  1:
Package Specifications And Bodies [SPC-88 5.6.2]
This compilation unit contains specification and body for the same package.
Separate them in different files

*** ERROR: STYLE VIOLATION Line  5, Column  6:
```

```
Types Integer, Natural and Positive [SPC-88 6.3.2]
Do not use predefined types (Integer, Natural and Positive)

*** ERROR: STYLE VIOLATION Line  8, Column  10:
Short Circuit Control Forms [SPC-88 4.2.1]
And not allowed. Use short-circuit forms instead (and then, etc.)

---------------------------------

 3 errors detected
 0 warnings detected
```

The option '**-v**' (verbose mode) specifies that we want output from the style checker on the standard output while it is executing.

Two files have now been created: A '**.vio**' file and a '**.sty**' file. The **.vio**-file contains a dump of the violations, corresponding to the output shown above, and the **.sty**-file is a copy of the original source code with the style violations inserted.

The file **ex1.sty** now has the following contents after the style checker has been activated:

```
package Example1 is
^
-- *** ERROR: STYLE VIOLATION Line  1, Column  1:
-- Package Specifications And Bodies [SPC-88 5.6.2]
-- This compilation unit contains specification and body for the same package.
-- Separate them in different files

end Example1;

package body Example1 is
  x: Integer;
-----^
-- *** ERROR: STYLE VIOLATION Line  5, Column  6:
-- Types Integer, Natural and Positive [SPC-88 6.3.2]
-- Do not use predefined types (Integer, Natural and Positive)

  b: Boolean;
begin
     b := b and (x = 0);
----------^
-- *** ERROR: STYLE VIOLATION Line  8, Column  10:
-- Short Circuit Control Forms [SPC-88 4.2.1]
-- And not allowed. Use short-circuit forms instead (and then, etc.)

end Example1;

---------------------------------

--  3 errors detected
--  0 warnings detected
```

Note that a reference to SPC's Ada Style Guide is provided for each style violation. For instance, the style rule causing the first style violation shown above is described in chapter 5.6.2 in the style guide.

Chapter 5

The Programmatic Interface

Apart from using the style checker as a stand-alone program, we have also provided a programmatic interface that can be used, for instance, in a software development environment.

The interface to such an environment takes the form of a package and consists of types, functions, and an iterator package.

The style checker package is with'ed in the compilation unit that intends to call the style checker.

The flow of control is as follows:

1. All the options to the style checker are specified.
2. The style checker is activated. It returns a list of style violations.
3. A dump of the violations or insertion of the violations in the source file can be performed (using the violation list and the functions described below).
4. Using the iterator package, violation records can be removed from the violation list one by one and the values of each component of the violation records (*e.g.*, the name of a violation) can be selected.

5.1 The Data Types

Options to the style checker are specified by providing the style checker procedure with an Ada record containing the following components:

- *Compilation_Unit_Only (type Boolean, default false):* Indicates whether the style checker should only perform the checks on the source file, or if the with'ed packages also should be checked.
- *Suppress_Explanation (type Boolean, default false):* Indicates whether the violation explanation should be omitted or not.
- *Log_To_Screen (type Boolean, default false):* Indicates whether the violations should be sent to the screen when checking.
- *Continue_On_Semantic_Error (type Boolean, default false):* Indicates whether the style checker should continue, even if semantic errors occur in the input. Note that if this component is set to true, the output of the style checker might not be reliable.

- *Suppress_vio_File (type Boolean, default false):* The style checker will automatically dump the violations in the file with '.vio' as the last name. This slot indicates whether this dump should be suppressed or not.
- *Stop_At_Violation_Number (type Natural, default 0):* If this number is greater than zero the style checker will stop before it reaches the end of the source file, after this number of style violations has been detected.

The types defined in the visible part of the programmatic interface are shown below:

```
package Style_Checker is

    type Option_Record is record
        Compilation_Unit_Only,
        Suppress_Explanation,
        Log_To_Screen,
        Continue_On_Semantic_Error,
        Suppress_vio_File: Boolean;
        Stop_At_Violation_Number: Natural;
    end record;

    subtype Line_Number_Type is Positive;
    subtype Column_Number_Type is Positive;
    type Name_Type is access String;
    type Explanation_Type is access String;
    subtype Warning_Type is Boolean;

    type Violation_Type is private;
    type Violation_List is private;

    :
end Style_Checker;
```

5.2 The Visible Operations

The programmatic interface consists of the following three operations:

- *Procedure Check:* This procedure performs the actual style checking. It is provided a file name and a data structure containing the options for style checking (*e.g.*, whether or not the style checker should continue on a semantic error, whether output should be printed on the screen, etc.). After the style checking has completed, a list of violations is returned together with a boolean variable that indicates whether the style checking was successful or aborted.
- *Procedure Dump:* This function dumps the violations into a file. The list of violations and the name of the list file are provided as an argument. Furthermore, the user can also indicate to the function, whether or not the violation explanation should be suppressed in the file.
- *Procedure Merge:* Provided with a list of violations, the name of the source file and the name of the output file, this procedure inserts the violations properly in the source file code and dumps the result in the list file specified. For each violation, a pointer to the exact place where the style violation has occurred is also provided.

The functions are specified as follows in the visible part of the style checker:

```
package Style_Checker is
⋮
   procedure Check(file: String; Options: Option_Record;
                                 Violations: out Violation_List;
                                 Success: out Boolean);

   procedure Dump(Violations: Violation_List; Log: in String;
                              Suppress_Explanation: Boolean);

   procedure Merge(Violations: Violation_List; Source, List: String);
⋮
end Style_Checker;
```

5.3 The Iterator Package

The visible part of the style checker also contains the specification for an iterator over violation lists. The following functions are provided to the user:

- *Function First_Violation:* Provided with a violation list, this function will return the first violation in the list. Furthermore, if a name is also provided, the function will return the first violation in the list that has this name.
- *Function Next_Violation:* Returns the next violation in the violation list. If a name is provided as an argument, the function will return the next violation in the violation list with this name.
- *Function Line_Number:* Given a violation, this function returns the line number where the style violation occurred in the source file.
- *Function Column_Number:* Given a violation, this function returns the column number where the style violation occurred in the source file.
- *Function Name:* Given a violation, this function returns the name of the style violation.
- *Function Explanation:* Given a violation, this function returns the explanation of why the style violation occurred.
- *Function Warning:* Returns True if the violation is a warning, and False if it is an error.
- *Exception No_More_Violation.* This exception is raised if First_Violation is provided an empty list or if Next_Violations reaches the end of the violation list.

The package is defined as follows in the style checker:

```
package Style_Checker is
⋮
   package Iterator is

      function First_Violation(Violations: Violation_List) return Violation_Type;
      function First_Violation(Violations: Violation_List; Name: Name_Type)
      return Violation_Type;
```

```
      function Next_Violation return Violation_Type;
      function Next_Violation(Name: Name_Type) return Violation_Type;

      function Line_Number(Violation: Violation_Type) return Line_Number_Type;
      function Column_Number(Violation: Violation_Type) return Column_Number_Type;
      function Name(Violation: Violation_Type) return Name_Type;
      function Explanation(Violation: Violation_Type) return Explanation_Type;
      function Warning(Violation:Violation_Type) return Warning_Type;

      No_More_Violations: exception;

   end Iterator;
   :
end Style_Checker;
```

Note that the actual visible part of the style checker contains more types and functions than described here. These are, however, only for internal use in the style checker and should be ignored. A failure to comply with this rule might cause an internal error in the style checker!

5.4 Putting It All Together

We now provide a simple example of how to use the style checker.

```
with Style_Checker;
use Style_Checker;

procedure Style_Driver is
   :
   -- We first declare the variables needed.
   vl: Violation_List;
   v: Violation_Type;
   o_r: Option_Record;
   s: Boolean;
   n: Name_Type;
   e: Explanation_Type;
   :
begin
   :
   o_r := (Compilation_Unit_Only => True,
           Suppress_Explanation => False,
           Log_To_Screen => True,
           Continue_On_Semantic_Error => False,
           Suppress_vio_File => False,
           Stop_At_Violation_Number => 0);
   -- O_r is the option record, which specifies the set of options, that the style checker should use.
   Check("test.a", o_r, vl, s);
   -- The style checker is activated.
   :
   Merge(vl, "test.a", "list_test.a");
```

```
   -- The style violations are inserted in the source code.
   :
   v := Iterator.First_Violation(vl);
   -- V holds the first violation.
   n := Iterator.Name(v);
   -- N holds the name/ID of the violation.
   e := Iterator.Explanation(v);
   -- E holds the explanation.
   :
end Style_Driver;
```

Chapter 6

Future Work

The current version, as it is presented in this report, is not fully implemented. More work has to be done in order to complete the system.

The following chapter lists, in order of priority, the different areas where future work would be beneficial.

6.1 The Style Checker Generator

The style checker is working, using the rules that we have included and supported in the Stanford version of the vocabulary package. However, the style checker generator, which should be able to produce the style rules package based on style rules specified in SSL, is not yet implemented.

Using the syntax described in Section 2, ALEX can be used to generate a parser for the style specification language.

AYACC can then be used to generate the style checker generator, such that SSL style rules directly can be translated into Anna annotations. The output of the style checker generator should be in the file named `'style_rules.Anna'`. The package in this file contains the **Style_Rules** package, which contains the 2 files: **Init_Style_Rules** and **Apply_Style_Rules**.

Examples of how conversion of SSL statements to Anna annotations should be performed is given in appendix A.

6.2 Using the Mitre Primitives

As described in the systems overview chapter, because the time frame for the style checker project has been relatively narrow, it has been difficult to apply the division of the vocabulary package into several abstraction layers. However, research at Mitre has produced an Ada query language described in [Byr89], which includes an implementation of 350 primitives to describe Ada programs. Including these primitives as lower level operations in the style checker would allow the functions available to the system manager to use these lower level functions, and thereby enforce the division of the vocabulary package into different layers of abstraction.

However, this will require a modification of the higher level functions, since they do not make use of these primitives.

6.3 Turning Style Rules Off

Since the user might not want all of the style rules to be invoked, when the style checker is activated, it would be convenient for the user, if the system allowed him to specify the rules, which should be turned off. This should include both the command line interface and the programmatic interface.

In order to implement this feature in the programmatic interface, another list has to be declared. This list could hold the style rule IDs for the style rules, that the user would like to deactivate.

The command line interface should allow the user to either specify the number of the style rule or the ID at the command line.

6.4 With'ed Packages

An Ada compilation unit can refer to other packages by including a 'with'-statement. The current version of the style checker only checks the input program specified by the user. However, future versions of the style checker should support the possibility of checking all with'ed compilation units, before the input unit is checked.

Switches for the command line interface and arguments for the programmatic interface are already provided in the current version.

6.5 X-Windows Environment

Providing the user with an X-Windows Environment would greatly enhance the user-friendliness of the system. The environment could allow incremental style checking, and on-line changes of the Ada program.

6.6 Using Webster's Dictionary

By providing an interface to the on-line Webster dictionary, the system manager will be able to write style specifications that pertain to the English-ness of names used in Ada programs. For example, the system manager may specify that portions of names between underscores have to be English words.

Bibliography

[Ada83] *The Ada Programming Language Reference Manual.* US Department of Defense, US Government Printing Office, February 1983. ANSI/MIL-STD-1815A-1983.

[Byr89] C. M. Byrnes. A DIANA query language for the analysis of Ada software. The Mitre Corporation, 1989.

[EBGW83] A. Evans, K. J. Butler, G. Goos, and W. A. Wulf. *DIANA Reference Manual, Revision 3.* Tartan Laboratories, Inc., Pittsburgh, PA, 1983.

[LvHBO87] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA, A Language for Annotating Ada Programs.* Volume 260 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987.

[San89] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN–CS–89–1282, and Computer Systems Laboratory Technical Report No. CSL–TR–89–391.

[SC88] M. Skalko and P. Cohen. Ada style guide, Software Productivity Consortium. SPC-TR-88-003, 1988.

Appendix A

The Stanford Style Checker

A.1 Implemented rules

The current version of the style checker checks for style violations according to the style rules (specified in English) shown in the following. The numbers correspond to the section numbering in SPC's Ada Style Guideline.

2.3.1: **Associate names with loops when they are deeply nested.** Names must be associated with loops which are nested more than 3 times.

2.3.6: **Use named parameter association in a subprogram call**

2.3.7: **Use default parameters when expanding or reducing functionality.** Actually, all subprogram parameters are required to have default value.

3.3.3: **Use derived types to re-use building blocks and enhance program maintainability.**

3.3.5: **Use private types in preference to explicitly exported types.** The kind of each type declaration is checked. Whenever it is not declaration of a private type a warning is issued.

3.4.5: **Restrict the depth of nested expressions and control structures.** Nesting of loops, if and case statements cannot be deeper than 3.

3.6.11: **Nest packages in the bodies not in the specification.**

3.7.2: **No declarations of objects in package specification.** Only subprograms, private types, (deferred) constants, and exceptions can be declared in the (visible part of) package specification. Note, that this rule is a stronger version of the rule shown in section 3.5.2.

3.7.3: **Minimize the number of the use clause.** Any occurrence of the "use clause" will cause violation of the above rule and will be reported.

3.7.7: **Do not include task entries in package specifications.** Any entry declaration appearing in package specification is reported as violation of the constraint.

4.2.1: **Use short-circuit forms**

4.3.1: **Create user-defined exceptions instead of explicitly raising predefined exceptions.** None of the explicitly raised exceptions may be predefined.

4.4.7: **Do not suppress exception checks.** Pragma "Suppress" is not allowed.

4.5.3: **Avoid aborting tasks.** Each abort statement will cause violation of the rule.

4.5.6: **Do not relay on the attributes "CALLABLE" or "TERMINATED".**

5.3.6: **Use shallow nesting of packages.**

5.6.2: **Always compile package specifications and bodies separately.**

6.3.2: **Do not use predefined types (INTEGER, NATURAL, POSITIVE).**

A.2 Rules Not Implemented

For various reasons (see below), the following rules have not been implemented so far. We have divided them up into 3 groups, depending on the reason for not implementing them.

Group I

2.1.1: **Use underscore to separate more than one word within a name.**

3.4.4: **Avoid names that rely on use of double negatives.**

Implementing these two rules would require access to a dictionary, which is not provided.

Group II

3.5.2: **Clarify large record structures by grouping related components into separate records.**

4.4.3: **Do not share variables.**

4.4.8: **Initialize all objects prior to use.** This requires data flow analysis, which might be time consuming to execute. Since the Verdix Ada compiler already performs this analysis, we have not given this rule a high priority.

5.5.5: **Force default initialization by using records for exported types.**

6.7.1: **Use the package facility to encapsulate implementation dependencies.**

6.11.2: **Do not use implementation-defined exceptions.**

6.13.4: **Avoid the use of package SYSTEM constants.**

6.14.1: **Avoid the use of additional I/O features provided by particular vendor.**

The rules in this have not been implemented merely because of the time constraints. They fall naturally within our paradigm of representing style guidelines as predicates on the language objects and do not require any additional resources. Few of the predicates and objects involved may present some problems but there seems to be no fundamental difficulty in implementing them within the current framework.

Group III

6.14.3: **Close all files explicitly.**

This rule can be verified only at run-time and, therefore requires a different approach to style checking. It is still easy to express it using Anna, *e.g.*, by means of objects like "file" and predicate "file_closed". But one would have to "modify" the program itself by introducing into it appropriate annotations. The Anna run time system would then check, while running the program, that eventually all files are being closed by explicit statements in the program. Since we did not consider annotating the checked ADA program, the rule has not been implemented.

A.3 Reference Guide

This section describes in details each implemented rule and shows its internal representation and intended SSL form.

Rule No. 2.3.1 - Warning

Guideline:

Associate names with loops when they are deeply nested.

SSL form:

```
For all x:loop :  self_nesting(x) > 3 -> named_stm(x);
```

Internal Representation:

```
--| object_OK(T, loop_ob) and then self_nesting(T) > 3 ->
--|       named_stm(T);
```

Description:

Signals violation whenever a loop statement is found which is nested more then twice, *e.g.*,

```
loop
    loop
        loop
```

where the outermost loop has no name. If only the inner loops have no names, *e.g.*,

```
Outer_Loop:
    loop
        loop
            loop
```

no violation is signaled.

Rule No. 2.3.6 - Error

Guideline:

Use named parameter association in a subprogram call

SSL form:

```
For all x:subprogram_call :  named_par_assoc(x);
```

Internal Representation:

```
--| object_OK(T,subprogram_call_ob) ->
--|       named_par_assoc(T);
```

Description:

"subprogram_call_ob" matches procedure and function calls but not entry calls. The predicate named_par_assoc(T) will, however, work also if T corresponds to an entry call. Therefore, if entry calls need be included in the scope of this rule one only has to add a new rule:

object_OK(T,entry_call_ob) – > named_par_assoc(T);

The predicate "named_par_assoc" returns true only if all parameters of the subprogram are named in the call.

Rule No. 2.3.7 - Warning

Guideline:

Use default parameters when expanding or reducing functionality.

SSL form:

```
For all x:in_parameter :  with_Default(x);
```

Internal Representation:

```
--| object_OK(T,in_parameter_ob) ->
--|       with_Default(T);
```

Description:

The rule affects the parameters of subprogram declarations, generic declarations, and entry declarations, but not record discriminants. In the example below, D1, Gg, and Gg1 will be reported as violating the rule.

```
procedure Def (D1: Integer; D2 : Integer := 2) is
    Gg: Integer;
    function Sq (Gg1: Integer) return Integer;
    :
```

Rule No. 3.3.3 - Warning

Guideline:

Use derived types to re-use building blocks and enhance program maintainability.

SSL form:

```
For all x:type_decl :
     is_a(x,private_type_decl) or
     is_a(x,incomplete_type_decl) or
     is_a(x,derived_type_decl);
```

Internal Representation:

```
--| object_OK(T, type_decl_ob) ->
```

```
--|       object_OK(T,private_type_decl_ob) or
--|       object_OK(T,incomplete_type_decl_ob) or
--|       object_OK(T,derived_type_decl_ob);
```

Description:

The kind of each type declaration is checked. Whenever it is not declaration of a derived type a warning is issued. Incomplete type declarations and declarations of the private types in the visible part of package specification are not reported.

Remarks:

The Style Checker is unable to find out the semantic information needed for deciding whether it is possible or reasonable to use a derived type instead of some other type. Therefore here, and in similar cases, only a warning is issued and it is up to the manager to make a decision concerning semantic reasons for using or not using derived type.

Rule No. 3.3.5 - Warning

Guideline:

Use private types in preference to explicitly exported types.

SSL form:

```
For all x:type_decl :  is_a(x,private_type_decl);
```

Internal Representation:

```
--| object_OK(T, type_decl_ob) ->
--|       object_OK(T, private_type_decl_ob) or is_within_private(T);
```

Description:

The kind of each type declaration is checked. Whenever it is not declaration of a private type a warning is issued. Note that not all type declarations occurring within the private part of a package are considered as declarations of private types. *E.g.*, in the following program both T2 and T4 will be classified as private types by the Checker but T3 will not.

```
package P1 is
    type T2 is private;
private
    type T2 is new Integer;
    package P2 is
        type T3 is new Integer;
        type T4 is private;
        F: Boolean;
    end P2;
end P1;
```

Rule No. 3.4.5 - Warning

Guideline:

Restrict the depth of nested expressions and control structures.

SSL form:

```
For all x:nested_stm :  logic_nesting(x) < 3;
```

Internal Representation:

```
--| object_OK(T, nested_stm_ob) ->
--|       logic_nesting(T, nested_stm_ob) < 3;
```

Description:

"nested_stm_ob" matches "if_ob", "case_ob" and "loop_ob". Consequently, the rule restricts mutual nesting of such statements to 3 levels.

```
while ... loop
    if ... then
        ... loop
            for ... loop
            end loop;
        end loop;
    else ...
    end if;
end loop;

while ... loop
    if ... then ... end if;
        for ... loop
            if ... then ... end if;
        end loop;
    if ... then ... end if;
end loop;
```

The first example illustrates a possible violation of the rule (logic_nesting=3). The second construct, however, will be legal (logic_nesting=2);

Remarks:

The predicate "logic_nesting" may be used for restricting nesting of particular statements . *E.g.*, "logic_nesting(T, if_ob)" will count only nesting of if-statements within the construct corresponding to T. Thus one might put restrictions on nesting of if-statements only by saying object_OK(T, if_ob) -> logic_nesting(T, if_ob) < 4; Inclusion of other statements (like for instance case-statement) into the category of "nested_stm_ob" is a straightforward extension. Nested expressions are not supported.

Rule No. 3.6.11/5.3.6 - Error

Guideline:

Nest packages in the bodies not in the specification. Use shallow nesting of packages.

SSL form:

```
For all x:package_decl :  count_all(x,package_decl) < 1;
For all y:package_body :  object_nesting(y,package_body) < 2;
```

Internal Representation:

```
--| object_OK(T, package_decl_ob) ->
--|       count_all(T,package_decl_ob) < 1;
```

```
--| object_OK(T,package_body_ob) ->
--|       object_nesting(T,package_body_ob) < 2;
```

Description:

For a given package specification T, counts all occurrences of package specifications within it. For a given package body T, counts the level of nesting of package bodies within it.

Rule No. 3.7.2 - Warning

Guideline:

No declarations of objects in package specification.

SSL form:

```
For all x:package_decl :
      count_1(visible_of(x), declaration) =
      count_1(visible_of(x), subprogram_decl) +
      count_1(visible_of(x), private_type_decl) +
      count_1(visible_of(x), exception_decl) +
      count_1(visible_of(x), deferred_constant_decl);
```

Internal Representation:

```
--| object_OK(T, package_decl_ob) or ->
--|       count_1(visible_of(T),declaration_ob) =
--|       count_1(visible_of(T),subprogram_decl_ob) +
--|       count_1(visible_of(x),private_type_decl) +
--|       count_1(visible_of(x), exception_decl_ob) +
--|       count_1(visible_of(x), deferred_constant_decl_ob);
```

Description:

"declaration_ob" matches declaration of exception, type, task, entry, variable, package, subprogram, subtype and constant. "object" in the Guideline refers to the static objects, *i.e.*, variable, package, type, entry, task, subtype, constant but we assume that it is usual to have declaration of exceptions and deferred constants in the visible part of package specification. The rule therefore says (SSL and internal form) that all declaration objects occurring in the visible part of the package spec should be declarations of subprograms, exceptions, private types or deferred constants. "visible_of" restricts the search space of the function "count_1" to the visible part of the package spec. (And the analogous function "private_of" returns the private part of package spec.) Thus all object declarations may be included in the private part of the package spec without violating the above rule. Using "count_1" makes sure that only first level declarations are counted. If a procedure declared within the package spec contains 10 declarations, these declarations will not affect the result of the above call to count_1.

Rule No. 3.7.3 - Warning

Guideline:

Minimize use of the use clause.

SSL form:

```
No (use);
```

Internal Representation:

```
--| not object_OK(T, use_ob);
```

Description:
Any occurrence of the "use clause" will cause violation of the above rule (the result will be FALSE) and will be reported.

Rule No. 3.7.7 - Error

Guideline:
Do not include task entries in package specifications.

SSL form:
```
For all x:package_decl:  count_all(x,entry_decl) = 0;
```

Internal Representation:
```
--| object_OK(T, package_decl_ob) ->
--|       count_all(entry_decl_ob) = 0;
```

Description:
Any entry declaration appearing in package specification is reported as violation of the constraint.

Rule No. 4.2.1 - Warning

Guideline:
Use short-circuit forms.

SSL form:
```
No (and); No (or);
```

Internal Representation:
```
--| not object_OK(T,and_ob);
--| not object_OK(T,or_ob);
```

Description:
"or_ob" and "and_ob" match all the application of functions "or" and "and". The short-circuit forms "and then" and "or else" are not such applications. Therefore, only the presence of the two former objects, "or" and "and", will be reported by the rules.

Rule No. 4.3.1 - Error

Guideline:
Create user-defined exceptions instead of explicitly raising predefined exceptions.

SSL form:
```
For all x:exception_raise :  not is_predefined(x);
```

Internal Representation:
```
--| object_OK(T, exception_raise_ob) ->
--|       not is_predefined(T);
```

Description:
The predicate "is_predefined" is true if its argument is declared in the package STANDARD. The rule says that none of the explicitly raised exceptions may be predefined.

Remarks:
One may still raise the predefined exceptions by means of the alternative "when others => raise". The predefined exceptions may also appear in the alternative choices of the exception handlers. The rule only ensures that a predefined exception is not raised by a raise statement.

Rule No. 4.4.7 - Error

Guideline:
Do not suppress exception checks.

SSL form:
```
No (pragma(SUPPRESS));
```

Internal Representation:
```
--| not object_OK(T, pragma_ob, "SUPPRESS");
```

Description:
Pragma "Suppress" is not allowed.

Remarks:
The third argument to the function "count_all" is the name provided for particular language objects like pragmas and attributes. It may be provided to any functions in the Vocabulary package which handle such language objects. Thus one may call count_all(T, attribute_ob, "CALLABLE") as well as count_all(T, attribute_ob, "TERMINATED").

One may also abstract from the name differences and call, *e.g.*, count_all(T, attribute_ob) if it is attribute as such, and not any particular attribute which counts.

Rule No. 4.5.3 - Warning

Guideline:
Avoid aborting tasks.

SSL form:
```
No (abort);
```

Internal Representation:
```
--| not object_OK(T, abort_ob);
```

Description:
Each abort statement will cause violation of the rule.

Rule No. 4.5.6 - Warning

Guideline:
Do not relay on the attributes "CALLABLE" or "TERMINATED".

SSL form:
```
No (attribute(CALLABLE)); No (attribute(TERMINATED));
```

Internal Representation:
```
--| not object_OK(T, attribute_ob, "CALLABLE");
--| not object_OK(T, attribute_ob, "TERMINATED");
```

Description:
Each use of attribute "Callable" ("Terminated") will be reported as a violation of the rule.

Rule No. 5.6.2 - Error

Guideline:
Always compile package specifications and bodies separately.

SSL form:
```
For all x:Compilation :  separate(x);
```

Internal Representation:
```
--| object_OK(T, main_ob) -> separate_comp(T);
```

Description:
Each compilation is checked for a package specification. (NB! Only compilation units of the given compilation are checked, *i.e.*, only package specifications at the outermost level are checked.) If, with exception of a package specification, there are other compilation units (not necessarily the body for the same package!) a violation is reported. Thus, the rule allows several subprograms or package bodies in one file but only one (and alone) package specification.
The same applies to generic packages.

Rule No. 6.3.2 - Error

Guideline:
Do not use predefined types (INTEGER, NATURAL, POSITIVE).

SSL form:
```
For all x:Any :  not refers_predefined(x,"INTEGER");
```

Internal Representation:
```
--| object_OK(T,any_ob) -> not refers_predefined(T,"INTEGER");
```

Description:
The rule forbids use of the above types by checking all the objects for their presence - whether as explicitly declared variable, parameters of the subprograms, result of functions, as array elements, in the declarations of other types, or when occurring implicitly in array range or for-loop iterations.

Remarks:
The predicate "refers_predefined" allows one to specify the type as the string argument. By default this argument is "INTEGER". The predicate then checks, for any node T, whether it, or any of its children which do not correspond to any actual language object (but is a merely structural node in DIANA), contain reference to

some type, the base type of which is_predefined (*i.e.*, defined in the package STANDARD; cf. rule 4.3.1). Since POSITIVE and NATURAL are subtypes of INTEGER, they will be included in the check for INTEGER. Notice that the rule does not preclude the use of the constants of the type Universal_Integer. This distinction is made by the Ada rules and we had to conform to it. Thus in the following program

```
function Def(D1: Integer) return Positive is
    Aa: array (1 .. 5) of Natural;
begin
    if 1 = 2 then ...
    elsif D1 = 3 then ...
    :
end Def;
```

Parameter D1 (both in the specification and when used in D1 = 3), result type (Positive), array element (Natural) as well as the array range (1 .. 5) will all be reported as violations. 1, 2, 3 after **begin**, however, will not because their base type is not predefined Integer, but Universal_Integer.

Appendix B

Maintenance Guide

The following describes the process of producing a new stand-alone version of the style checker. We assume that an Anna library has been created by invoking **Anna.mklib**, and that all the files necessary to create the style checker are present in the directory.

The Stanford version of the style checker is currently found in the directory:

`/anna/lockheed/style`

As described earlier, the system consists of many different packages, which are all linked to form a 1.8 to 2MB executable file (The Stanford version including 20 rules takes approximately 1.84MB in version 1.20).

The style checker is divided into several packages:

- **List_Package Package:** The list package specification is given in `list.Anna` and the body in `list_.Anna`.
- **Voc Package:** The package specification is found in `voc_v.Anna`, and the body in `voc_b.Anna`.
- **Style_Rules Package:** The package is found in `style_rules.Anna`.
- **Style_Checker Package:** The specification of the package is found in `style_v.Anna` and the body in `style_b.Anna`.
- **Style_Driver Procedure:** The package is found in `style_driver.Anna`.
- **Dummy Packages:** In order to run the Anna transformer, several dummy packages/stubs have been made to cover packages that the Anna transformer should not transform, *e.g.*, Text_IO, Command_Line, SYSTEM, etc. These dummy packages are found in the files `dummies1.Anna` and `dummies2.Anna`.

A 'makefile' has been made to ease the transformation, compiling and linking.

The following files are required to recompile the complete style checker from scratch: `list.Anna`, `list_.Anna`, `dummies1.Anna`, `dummies2.Anna`, `voc_v.Anna`, `voc_v.a`, `voc_b.Anna`, `voc_b.a`, `style_v.Anna`, `style_rules.Anna`, `style_b.Anna`, `style_driver.Anna`, the correct make file (`Makefile`).

Furthermore, all paths needed to execute the Ada compiler should be set up.

The following stages are needed in order to produce the executable file:

1. Transforming the Anna files into Ada files by running the Stanford Anna Transformer on the **.Anna** file. Note that this does not apply to voc_v.Anna and voc_b.Anna. These files are only used as dummy 'stubs', *i.e.*, they contain only the information necessary to make the Anna Transformer transform the subsequent packages correctly. **Thus, care should be taken to save a copy of the vocabulary package, since running the transformer causes voc_?.a to be overwritten.** We have saved a copy in the files voc_?.back.
2. Compiling each package using the Ada compiler. (Now remember to compile the correct copies of voc_v.a and voc_b.a.)
3. Repeating step 1 and 2 until all packages have been compiled, and then
4. Linking the packages and producing the executable file.

The end result is now an executable file, which can be activated from the command line (See Chapter 4).

The whole process of transforming, compiling, and linking is then invoked by issuing the following command line call:

```
make style
```

As shown above, the order of processing should be as follows:

1. `list.Anna`
2. `list_.Anna`
3. `dummies1.Anna`
4. `dummies2.Anna`
5. `voc_v.Anna`
6. `voc_b.Anna`
7. `voc_b.Anna`
8. `style_v.Anna`
9. `style_rules.Anna`
10. `style_b.Anna`
11. `style_driver.Anna`
12. Linking using `a.ld`

If only the programmatic interface is needed, the file **style_driver.Anna** is substituted with the new file and everything is compiled and linked as before.